DISCOVERING PERMUTATION PATTERNS

Field of the Invention

The present invention relates to pattern discovery and, more particularly, relates to discovery of permutation patterns.

Background of the Invention

A permutation pattern is a pattern where the characters in the pattern can be in any order. For instance, in the input string, $S = abc...cab$, a permutation pattern can be described as $\{a, b, c\}$ and the permutation pattern occurs at locations 1 and 7 in the input string. Permutation patterns have a variety of practical uses.

For example, genes that appear together consistently across genomes are believed to be functionally related: these genes in each other's neighborhood often code for proteins that interact with one another, suggesting a common functional association. However, the order of the genes in the chromosomes may not be the same. In other words, a group of genes appear in different permutations in the genomes. For example in plants, the majority of snoRNA genes are organized in polycistrons and transcribed as polycistronic precursor snoRNAs. Also, the olfactory receptor(OR)-gene superfamily is the largest in the mammalian genome. Several of the human OR genes appear in clusters, with ten or more members located on almost all human chromosomes. Furthermore, some chromosomes contain more than one cluster, where a cluster has one or more permutation patterns.

As the available number of complete genome sequences of organisms grows, it becomes a fertile ground for investigation along the direction of detecting gene clusters by comparative analysis of the genomes. A gene G is compared with its orthologs G' in the different organism genomes. Even phylogenetically close species are not immune from gene shuffling, such as in Haemophilus influenzae and Escherichia Coli. Also, a multicistronic gene cluster sometimes results from horizontal transfer between

species and multiple genes in a bacterial operon fuse into a single gene encoding multi-domain protein in eukaryotic genomes.

If the functions of genes, say G_1G_2, are known, the function of its corresponding ortholog clusters $G_2'G_1'$ may be predicted. Such positional correlation of genes as clusters and their corresponding orthologs have been used to predict functions of ABC transporters and other membrane proteins.

The local alignment of nucleic or amino acid sequences, called the multiple sequence alignment problem, is based on similar subsequences; however the local alignment of genomes is based on detecting locally conserved gene clusters. A measure of gene similarity is used to identify the gene orthologs. For example, genes $G_1G_2G_3$ may be aligned with $G_1'G_2'G_3'$, and such an alignment is never detected in subsequence alignments.

Domains are portions of the coding gene (or the translated amino acid sequences) that correspond to a functional sub-unit of the protein. Often, these are detectable by conserved nucleic acid sequences or amino acid sequences. The conservation helps in a relative easy detection by automatic motif discovery tools. However, the domains may appear in a different order in the distinct genes giving rise to distinct proteins. But, they are functionally related due to the common domains. Thus these represent functionally coupled genes such as forming operon structures for co-expression.

Thus, it can be seen that it would be useful to determine permutation patterns for genes or proteins. Consequently, there is a need for improved techniques for determining permutation patterns.

Summary of the Invention

The present invention provides techniques for determining permutation patterns.

In an exemplary aspect of the present invention, a new portion of an input string is selected. The input string has a number of characters from an alphabet. The new

portion differs from a previously selected portion of the input string by one or more new characters of the input string. One or more values are determined for how many of the one or more new characters are in the portion of the input string. It is determined which, if any, names in a number of sets of names have changed by selection of the new portion. The sets have a first set and a number of additional sets, wherein the first set corresponds to all of the characters in the alphabet and to values of how many of the characters of the alphabet are in the previously selected portion. The values are names for the first set. Each additional set comprises names corresponding to selected pairs of names from a single other set. Changes in the names are used to determine the permutation patterns.

Each name generally corresponds to a permutation pattern and permutation patterns may be found by keeping track of changes to the names. When a name is changed greater than or equal to a predetermined value, the permutation pattern corresponding to the name may be output.

A more complete understanding of the present invention, as well as further features and advantages of the present invention, will be obtained by reference to the following detailed description and drawings.

Brief Description of the Drawings

FIG. 1 shows an example of permutation patterns discovered in an exemplary input string;

FIG. 2 is an exemplary method for determining permutation patterns, in accordance with a preferred embodiment of the present invention;

FIGS. 3A and 3B are exemplary naming trees used to describe techniques of the present invention; and

FIG. 4 is an exemplary system for determining permutation patterns from an input string, in accordance with a preferred embodiment of the present invention.

Detailed Description of Preferred Embodiments

For ease of reference, the present disclosure is divided into the following sections: Introduction; The Permutation Pattern Problem; Maximal Patterns; and Techniques for Finding Permutation Patterns.

Introduction

The present invention allows permutation patterns to be discovered. In this disclosure, the abstract problem of discovering permutation patterns is formed as a discovery problem called the π pattern problem and techniques that automatically discover permutation patterns in, for instance, multiple input patterns are given. As there is generally not enough knowledge about forming an appropriate model to filter the meaningful from the apparently meaningless permutation patterns, a model-less approach is taken herein, which allows all permutation patterns that appear a number of times to be determined. Additionally, a notation is introduced for maximal permutation patterns that drastically reduces the number of valid cluster patterns, without any loss of information, making it easier to study the results from an application viewpoint.

The Permutation Pattern Problem

The permutation pattern problem is described below. A permutation pattern will sometimes be referred to through a "π pattern" shorthand.

The section begins by relating some definitions.

Let $S = s_1 s_2 ... s_n$ be a string of length n, and $P = p_1 p_2 ... p_m$ a pattern, both over alphabet $\{1, ..., |\sum|\}$.

Definition $(\Pi(s), \Pi'(s))$. Given a string s on alphabet $\sum$,

$\Pi(s) = \{a \in \sum | a = s[i], \text{for some } 1 \le i \le |s|\}$ and

$\Pi'(s) = \{a(t) | a \in \Pi(s), t \text{ is the number of times that } a \text{ appears in } s\}$

For example if *s=abcda*, $\Pi(s) = \{a, b, c, d\}$. As another example, if *s=abbccdac*, $\Pi'(s) = \{a(2), b(2), c(3), d\}$. Note that d appears only once and the annotations are ignored altogether.

Definition of "p-occurs." A pattern P p-occurs, which means that there is a permuted occurrence of the pattern, in a string S at location i if: $\Pi'(P) = \Pi'(s_i \ldots s_{i+m-1})$.

Definition of a permutation pattern, called a "π pattern." Given an integer K, a pattern P is a π pattern on S if:

$|P| > 1$, where this step rules out trivial single character patterns; and

P p-occurs at some $k' \geq K$ distinct locations on S. $\pounds_p = \{i_1, i_2, \ldots, i_k\}$ is the location list of p.

For example, consider $\Pi'(P) = \{a(2), b(3), c(3)\}$, and the string S=*aacbbxxabcbab*. Clearly, P p-occurs at positions 1 and 9.

A problem of π pattern discovery. Given a string S and $K < n$, find all π patterns of S together with their location lists.

For example, if S=*abcdbacdabacb*, then $P = \{a, b, c\}$ is a 4-π pattern with location list $\pounds_p = \{1, 5, 10, 11\}$.

The total number of π patterns is $O(n^2)$, but is this number actually attained? Consider the following example.

Let S= *abcdefghijabdcefhgij* and $k = 2$. The π patterns shown in FIG. 1 show that their number could be quadratic in the size of the input.

Maximal Patterns

A general definition is given below of maximality, and this definition holds even for different kinds of substring patterns such as rigid, flexible, with or without wild cards. Maximal patterns are also described in Parida, "Some Results on Flexible-Pattern Matching," Proc. of the 11th Symp. on Comp. Pattern Matching, vol. 1848 of Lecture Notes in Comp. Sci., 33-45 (2000), the disclosure of which is hereby incorporated by reference.

In the following, assume that $\wp$ is the set of all π patterns on a given input string S.

Definitions of "non-maximal" and "maximal" patterns are as follows. $P_a \in \wp$ is non-maximal if there exists $P_b \in \wp$ such that: (1) each p-occurrence of P_a on S is covered by a p-occurrence of P_b on S (each occurrence of P_a is a substring in an occurrence of P_b) and, (2) each p-occurrence of P_b on S covers $l \geq 1$, p-occurrence(s) of P_a on S. A pattern P_b that is not non-maximal is maximal.

Clearly, $\Pi'(P_a) \subset \Pi'(P_b)$. Although it seems counter-intuitive, it is possible that $|£_{pa}| < |£_{pb}|$. Consider the input S = *abcdebca.....abcde*. $P_a = \{d, e\}$ p-occurs only two times but $P_b = \{a, b, c, d, e\}$ p-occurs three times and by the definition P_a is non-maximal with respect to P_b.

To illustrate the case of $l > 1$in the definition, consider

S= abcdbac.....abcabcd.....abcdabc.

$P_a = \{a, b, c\}$ p-occurs two times in the first and third, and, four times in the second p-occurrence of $P_b = \{(a)2, (b)2, (c)2, d\}$. Also, by the definition, P_a is non-maximal with respect to P_b.

It is also claimed that such a non-maximal pattern P_a can be "deduced" from P_b and the p-occurrences of P_a on S can be estimated to be within the p-occurrences of P_b. This is shown in more detail below.

The following can be shown. Let $M = \{P_j \in \wp | P_j$ is maximal$\}$. M is unique.

This is straightforward to see. This result holds even when the patterns are substring patterns.

In example shown in FIG. 1, pattern P_7 is the only maximal π pattern in S.

Maximality notation will now be described. Recall that in case of substring patterns, the maximal pattern very obviously indicates the non-maximal patterns as well. For example, a maximal pattern of the form *abcd* implicates *ab, bc, cd, abc, bcd* as possible non-maximal patterns, unless they have occurrences not covered by *abcd*. Do maximal π patterns have such an obvious form? In this section, a special notation is

introduced based on observations discussed below. It is then demonstrated how this notation makes it possible to represent maximal π patterns.

Let $Q \in \wp$ and $\partial = \{Q' | Q'$ is non-maximal w.r.t. $Q\}$. Then there exists a permutation, $\overline{Q}$, of $\Pi'(Q)$ such that for each element $Q' \in \partial$, a permutation of $\Pi'(Q')$ is a substring of $\overline{Q}$.

Without loss of generality, let the ordering of the elements be as the one in the leftmost occurrence of Q on S as $\overline{Q}$. Clearly, there is a permutation of $\Pi'(Q')$ that is a substring of $\overline{Q}$, else Q' is not a non-maximal pattern by the definition.

The ordering is not necessarily complete. Some elements may have no order with respect to some others.

Consider S = *abcdef.....cadbfe.....abcdef.* Then $P_1 = \{a, b, c, d\}$, $P_2 = \{e, f\}$ and $P_3 = \{a, b, c, d, e, f\}$ are the π patterns with three occurrences each on S. Then the intervals denoted by brackets can be represented as

$$({}_3({}_1a, b, c, d)_1, ({}_2e, f)_2)_3,$$

where the elements within the brackets can be in any order. A pair of brackets $({}_i...)_i$ corresponds to the π pattern P_i. An element is either a character from the alphabet or bracketed elements.

A representation that captures the order of the elements of Q along with the intervals that correspond to each Q' encodes the entire set Q. This representation will appropriately annotate the ordering. The representation using brackets works except that there may intersecting intervals that could lead to clutter. When the intervals intersect, the brackets need to be annotated. For example, *(a(b,d)c)* can have at least two distinct interpretations: (1) $({}_1a({}_2b, d)_2c)_1$, or, (2) $({}_1a({}_2b, d)_1c)_2$.

Consider the input string S = *abcd.....dcba.....abcd.* The π patterns are $P_1 = ab$, $P_2 = bc$, $p_3 = cd$, $P_4 = abc$, $P_5 = bcd$, $P_6 = abcd$, each occurring three times. Using only annotated brackets will yield a cluttered representation as follows:

$$({}_6({}_1({}_4a({}_2({}_5b)_1({}_3c)_2)_4d)_3)_5)_6.$$

The annotation of the brackets is beneficial to keep the pairing of the brackets unambiguous. It is clear that if two intervals intersect, then the intersection elements are immediate neighbors of the remaining elements. For example, if ${}_1a({}_2b, c)_1d)_2$, then *(b,c)* must be immediate neighbors of (*a*) as well as (*d*). If a symbol '–' is introduced to denote immediate neighbors, then the intervals never intersect. Further, they do not need to be annotated if they do not intersect. Thus the previous example can be simply given as *a-(b,c)-d*. The earlier cluttered representation can be cleanly put as the following:

a-b-c-d.

Next, consider the example shown in FIG. 1. Using the notation, there is only one maximal π pattern given by *M=a-b-(c,d)-e-f-(g,h)-i-j* at locations 1 and 11 on *S*. Notice that $\Pi(P_7) = \Pi(M)$ and every other π pattern can be deduced from *M*.

Techniques for Finding Permutation Patterns

When finding patterns, the input is generally a set of strings of total length *n*. In order to simplify the explanation, one string *S* of length *n* over an alphabet $\sum$ will be considered. It should also be noted that each string can comprise sets of characters at each location in the string. However, for simplicity, one character per location will be described herein.

The techniques presented below compute the maximal π patterns in *S*. The maximal π patterns can be determined in two stages: (1) find all the π patterns in *S*, and (2) find the maximal π patterns in *S*. In an exemplary implementation, in Stage 2, a straightforward computation is used that uses location lists of all the π patterns in *S* obtained at Stage 1. The location lists of each pair of π patterns are checked to find if one π pattern is covered by another one. Assume that Stage 1 outputs p π patterns, and the maximum length of a location list is l, Stage 2 runs in $O(p^2 l)$ time. From now on, only Stage 1 will be discussed.

It is assumed that the size of the longest pattern is L. Step l of Stage 1, where $2 \leq l \leq L$, finds π patterns of length l. Stage 1 is described broadly by the method of FIG. 2. Steps of method 100 will be described broadly, then more detailed description of the steps will be given.

Method 100 of FIG. 2 selects a window size (step 110), and then moves a window of size l along string S, adding and deleting a letter in each iteration. In step 115, the window 170 is placed at the beginning of the string S, as shown by reference 155. In this example, the window size, l, is four.

A naming tree, described in detail below, is updated in step 120. It should be noted that this step can include determining new names. In step 125, a search is made for updated names in the naming tree. It is this search that lessens the time spent while determining π patterns. Counters are updated in step 130 for the updated names. This step may also comprise updating location lists. In step 135, it is determined if the end of the string has been reached. If it has not (step 135 = NO), then the window 170 is moved one character to the right (in this example), as shown by reference 160. Method 100 continues until the end of the string is reached (step 135 = YES), when the window size is changed in step 140. If the window size is not greater than the size of the string (step 145 = NO), method 100 continues in step 110. If the window size is greater than the size of the string, the method ends in step 150, where the patterns that appear greater than K times are output as permutation patterns.

Method 100 will now be described in more detail.

The method 100 maintains, in step 120 for instance, an array NAME $[1...|\sum|]$ where NAME*[q]* keeps count of the number of appearances of letter q in the current window. Hence, the sum of the values of the elements of the NAME array is l. In each iteration the window shifts one letter to the right, and at most 2 variables of the NAME array are changed: one is increased by one (e.g., adding the rightmost letter) and one is decreased by one (e.g., deleting the leftmost letter of the previous window).

Note that for a given window $s_a s_a + 1...s_{a+l-1}$ the NAME array represents $\Pi'(s_a s_{a+1}...s_{a+l-1})$. There is one difference between the NAME array and Π', and that is

that in Π' only the letters of Π are considered and, in the NAME array, all letters of Σ are considered, but the values of letters that are not in Π are zero. At iteration j of steps 115 through 135, the NAME array is defined to represent the substring $s_j...s_{j+l-1}$.

An observation can be made that substrings of S, of length l, that are permutations of the same string are represented by the same array NAME.

It has been explained how the NAME arrays of all substrings of length l of S are computed. The NAME arrays that appear more than K times still need to be found.

In an embodiment of the present invention, each distinct NAME array is given a unique name, which is an integer in the range $0...n$. The choice of assigning an integer is arbitrary, as any code could be used for a name. The names are given by using the naming technique, described below.

A suitable naming technique is described as follows. Assume, for the sake of simplicity, that $|\Sigma|$ is a power of 2. (If $|\Sigma|$ is not a power of 2, the NAME array can be extended to an appropriate size by concatenating to its end repeated -1. The size of the resulting array is no more than twice the size of the original array.)

A name is given to each subarray of size 2^i that starts on a position $j2^i + 1$ in the array, where $0 \le i \le \log|\Sigma|$ and $0 \le j \le |\Sigma|/2^i$. Names are given first to subarrays of size 1 then $2,4,...|\Sigma|$, at the end a name is given to the entire array.

A subarray of size 2^i is a concatenation of 2 subarrays of size 2^{i-1}. The names of these 2 subarrays are used as the input for the computation of the name of the subarray of size 2^i. The process may be viewed as constructing a naming tree, which can be considered, in an exemplary embodiment, to be a binary tree. The naming tree has a number of levels. The leaves of the tree (e.g., at level 0, as shown below) are the elements of the initial array. Node x in level i is the parent of nodes $2x-1$ and $2x$ in level $i-1$.

An exemplary naming strategy is as follows. A name is a pair of previous names. At level j of the naming, we compute the name of subarray $NAME_1NAME_2$ of size 2^j, where $NAME_1$ and $NAME_2$ are consecutive subarrays of size 2^{j-1} each. In an exemplary embodiment, names are given as natural numbers in increasing order. Notice

that every level only uses the names of the level below it, thus the names used at every level are numbers from the set *{1,...,n}*.

To give an array a name, it is only necessary to know if the pair of names of the composing subarrays has appeared previously. If it did, then the array gets the name of this pair. Otherwise, it gets a new name. It is necessary, therefore, to show a quick way to dynamically access pairs of numbers from a bounded range universe.

An example will help to explain this. Let the alphabet be as follows: $\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}, |\Sigma| = 16$. Assume a substring *cboljikgikl* of *S*, the NAME that represents this substring is as shown in FIG. 3A. The term NAME refers to row 250 and each entry in rows 210-240. Each name also represents a pattern, which will be used to determine permutation patterns (e.g., patterns that occur $\geq K$ times). Additionally, the rows 210-250 can be considered sets of names. The row 250 has the leaves, and each entry 250-1 through 250-16 corresponds to a character of the alphabet. For instance, entry 250-1 has a value of zero and corresponds to the number of characters "*a*" there are in the substring. Entry 250-2 has a value of one and corresponds to the number of characters "*b*" there are in the substring. Similary, entry 250-9 has a value of two and corresponds to the number of characters "*i*" there are in the substring, while entry 250-16 has a value of zero and corresponds to the number of characters "*p*" there are in the substring. The entry 240-1 is a name assigned to the value "01" from the pair 250-1 and 250-2. Similarly, the entry 230-1 is a name assigned to the value "43" from the pair 240-1 and 240-2.

Suppose the window move adds the character *n*. (It should be noted that no character drops off in this example; instead the window grows in size to encompass the character *n*.) In the diagram shown in FIG. 3B, the names that changed as a result of the change to the naming tree are shown in shading.

From this example, one can see that a single change in the array NAME causes at most $|\Sigma|$ names to change, since there is at most one name change in every level 210 through 250.

It can be shown that at every iteration, only $O(\log|\Sigma|)$ names need to be handled, since only two elements of array NAME are changed.

It has been shown that the name of the NAME array can be maintained at a cost of $O(\log|\Sigma|)$ per iteration. What should be found is whether the updated NAME array gets a new name, or a name that appeared previously. Before an efficient implementation of this task is shown, the maximum number is bound for different names needed to generate for a fixed window size l.

It can be shown that the maximum number of different names generated by the techniques of the present invention's naming of size l window on a text of length n is $O(n\log|\Sigma|)$. The maximum number of names generated at a fixed level j in the naming tree is $O(n)$.

A pair recognition problem is now discussed. It was shown earlier that it is beneficial to show a quick way to dynamically access pairs of numbers from a bounded range universe. Formally, a solution to the following problem is to be found:

The dynamic pair recognition problem is the following:

INPUT: A sequence of queries $\{(a_j, b_j)\}_{j=1}^{\infty}$ where $a_,, b_j \in \{1, ..., j\}$.

OUTPUT: Dynamically decide, for every query (a_j, b_j), whether there exist c, $c<i$ such that $(a_j, b_j) = (a_c, b_c)$.

At any point j, the pairs being considered all have their first element no greater than j. Thus, accessing the first element can be done in constant time by direct access. This suggests "gathering" all pairs in trees rooted at their first element. However, if it can be assured that these trees are ordered by the second element and balanced, elements can be found by binary search in time that is logarithmic in the tree size.

The above solution, for the pair recognition problem, can be determined, when solving each query (a_j, b_j), through a search on a balanced search tree with all previous queries whose first pair element is a_j. Since in every level there are at most $O(n)$ different numbers, the time for searching such a balanced search tree is $O(\log|BAL[a]|) = O(\log(n))$. Balanced search trees are described in Introduction to

Algorithms, T. Cormen, C. Leiserson, and R. Rivest, MIT Press, 381-399 (1991), the disclosure of which is hereby incorporated by reference.

The above technique gives names to many parts of NAME. Special attention is given to leaves, in the balanced search trees, that represent names of the entire array NAME. In other words, a leaf could represent one row 250 of FIG. 3. In Step l of method 100 of FIG. 2, all the patterns of one π pattern will reach the same leaf. A counter can be added to each leaf that finds if the number of occurrences is at least K. Additionally, a location list can be added to each leaf.

The time complexity of Stage 1, method 100 of FIG. 1, may be computed as follows. Stage 1 runs L times. In a step l, NAME and the naming tree are initialized in $O(l+|\Sigma|)$ time and then compute n-l iterations are computed. Each iteration includes at most two changes in NAME, and the computation of $O(\log|\Sigma|)$ names. Computing a name takes $O(\log n)$ time. Hence the total running time of Stages 1 and 2 is $O(Ln\log|\Sigma|\log n)$.

Turning now to FIG. 4, an exemplary computer system 300 is shown for determining permutation patterns, in this example maximal permutation patterns 340, from an input string 340. Computer system 300 comprises a processor 310 coupled to memory 315, which comprises a pattern discovery process 320, a naming tree 325, counters 330, location lists 335, and database 337. The pattern discovery process 320 takes one or more input strings 305 and determines maximal permutation patterns 340, as described above. The pattern discovery process 320 performs Stages 1 and 2 as described above. In the example of FIG. 1, the counters are used to store how many times each named pattern occurs and are stored separately, in this example, from the location lists 335 and naming tree 325. The location lists 335 tell where the patterns occur. As described above, one way to implement naming tree 325, counters 330, and location lists 335 is through a balanced search tree 345. Balanced search tree 345 comprises a number of nodes 350-1 through 350-5, of which nodes 350-2, 350-3, and 350-5 are leaves.

Database 337 may be used to store results from previous calculations using the present invention. Database 337 may be used, for instance, in the following manners.

Given a database 337, D, and a query sequence s, then D may be used to check how similar s is to zero elements, one element or several elements in D. The similarity could be in terms of "local composition." This translates to finding permutation patterns that are common to s and D. Then the techniques of the present invention may be used to detect these regions of similarity.

The present invention described herein may be implemented as an article of manufacture comprising a machine-readable medium, as part of memory 315 for example, containing one or more programs that when executed implement embodiments of the present invention. For instance, the machine-readable medium may contain a program configured to perform steps in order to perform Stages 1 and 2 described above. The machine-readable medium may be, for instance, a recordable medium such as a hard drive, an optical or magnetic disk, an electronic memory, or other storage device.

It is to be understood that the embodiments and variations shown and described herein are merely illustrative of the principles of this invention and that various modifications may be implemented by those skilled in the art without departing from the scope and spirit of the invention.